EVORAM: A SELF-EVOLVING EMBODIED AGENT THROUGH EXECUTABLE MEMORY ACCUMULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Embodied agents are increasingly capable of solving complex robotic tasks through powerful foundation models and diverse tools, yet they often approach new problems from scratch, with successful exploration providing limited benefit to future interactions. We argue that a self-evolving embodied agent should instead continuously transform successful exploration into persistent experience and leverage accumulated experience to facilitate subsequent exploration. To this end, we introduce EVORAM (**Evo**lving **R**obotic **A**gent with **M**emory), a self-evolving embodied agent built around executable memory accumulation. EVORAM combines a VLA loop for flexible and reactive interaction with a code loop for deliberate exploration and reproducible execution. Successful solutions discovered through exploration are physically verified and consolidated into structured executable memory, which can be directly reused, adapted, and further refined when solving subsequent problems. The two loops further complement each other: diverse VLA behaviors broaden code-based exploration, while verified code provides reliable recovery from VLA failures. Experiments in simulation and on a real-world dual-arm robot demonstrate this self-evolution process. On LIBERO, EVORAM progressively acquires, reuses, and refines executable experience, which in turn enhances subsequent exploration, ultimately enabling it to solve 121/130 (93.1%) tasks. Real-world experiments further demonstrate that such executable experience can generalize to novel but related objects, while Code–VLA collaboration broadens the exploration space and accelerates the accumulation of executable memory. Remarkably, after acquiring the relevant memory, EVORAM can autonomously calibrate its camera intrinsics and camera–robot extrinsics, enabling sustained exploration in the physical world despite calibration degradation. These results demonstrate a self-evolution loop in which exploration continuously expands executable memory, while accumulated memory in turn strengthens subsequent exploration.

1 INTRODUCTION

Recent advances in embodied intelligence have revealed two powerful paradigms for building general-purpose robotic agents. Code-based agents leverage the reasoning capabilities of large language models (LLMs) to decompose tasks, invoke tools, and generate executable solutions (Liang et al., 2023; Singh et al., 2023; Huang et al., 2023b;a; Vemprala et al., 2023), making them suitable for deliberate exploration of novel problems. In contrast, vision-language-action (VLA) models directly map multimodal observations and language instructions to robot actions (Brohan et al., 2023; Kim et al., 2024; Black et al., 2026; Team et al., 2024; O'Neill et al., 2025), enabling efficient and reactive control. However, both paradigms lack mechanisms to persistently consolidate successful interactions during deployment into reusable experience, limiting their ability to continuously benefit from past interactions when solving new problems. This raises a fundamental question: **can an embodied agent continuously turn successful exploration into persistent experience that strengthens subsequent exploration?** Such a process would form a self-evolution loop, where exploration expands memory and accumulated memory in turn strengthens exploration.

However, realizing this vision presents two key challenges. First, how can an embodied agent continuously explore previously unsolved tasks and acquire new executable solutions through interaction (**Exploration**)? Recent systems such as ETA (Chen et al., 2026) and ENPIRE (Xiao et al.,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2026) have taken initial steps toward autonomous improvement, but either struggle to consistently benefit from accumulated experience or focus primarily on optimizing individual tasks. Harness VLA (Zhang et al., 2026) combines a coding agent with VLA control to improve reliability in complex long-horizon tasks, but does not fully exploit their complementary capabilities for open-ended exploration. Second, how can successful exploration be persistently consolidated and reused to strengthen subsequent exploration (**Memory**)? EmbodiedCoder (Lin et al., 2025) and ASPIRE (Lu et al., 2026) demonstrate the benefits of code or skill reuse, yet scalable long-term memory remains unresolved. Analytic Concept-Centric Memory (Sun et al., 2026) provides more structured experience organization, but relies on fixed semantic templates and manipulation primitives, limiting the continual expansion of executable experience.

To address these challenges, we propose **EvoRAM** (**Evo**lving **R**obotic **A**gent with **M**emory), a self-evolving embodied agent that continuously explores, accumulates, and reuses executable experience. EvoRAM combines code-based reasoning with VLA execution to broaden exploration beyond their individual capabilities, while successful solutions are physically verified and consolidated into persistent executable memory. The accumulated memory can be directly reused or adapted to facilitate subsequent exploration. In turn, continued exploration expands and refines the memory. Through this mutually reinforcing loop between exploration and memory accumulation, EvoRAM progressively acquires new capabilities through interaction.

Our main contributions are summarized as follows:

- **Self-Evolving Embodied Agent.** We introduce EvoRAM, a self-evolving agent that integrates code-based and VLA exploration. It forms a mutually reinforcing loop where exploration expands executable memory, while accumulated experience strengthens subsequent exploration.
- **Persistent Executable Memory.** We propose a structured memory that consolidates verified experience across objects, tasks, and planning into executable knowledge that can be directly reused, adapted, and refined for subsequent exploration.
- **Extensive Simulation and Real-World Evaluation.** We validate the self-evolution loop in both simulated and real-world environments. In simulation, EvoRAM iteratively explores and eventually solves 121/130 (93.1%) LIBERO tasks. In the physical world, accumulated memory enables EvoRAM to autonomously complete camera–robot recalibration.

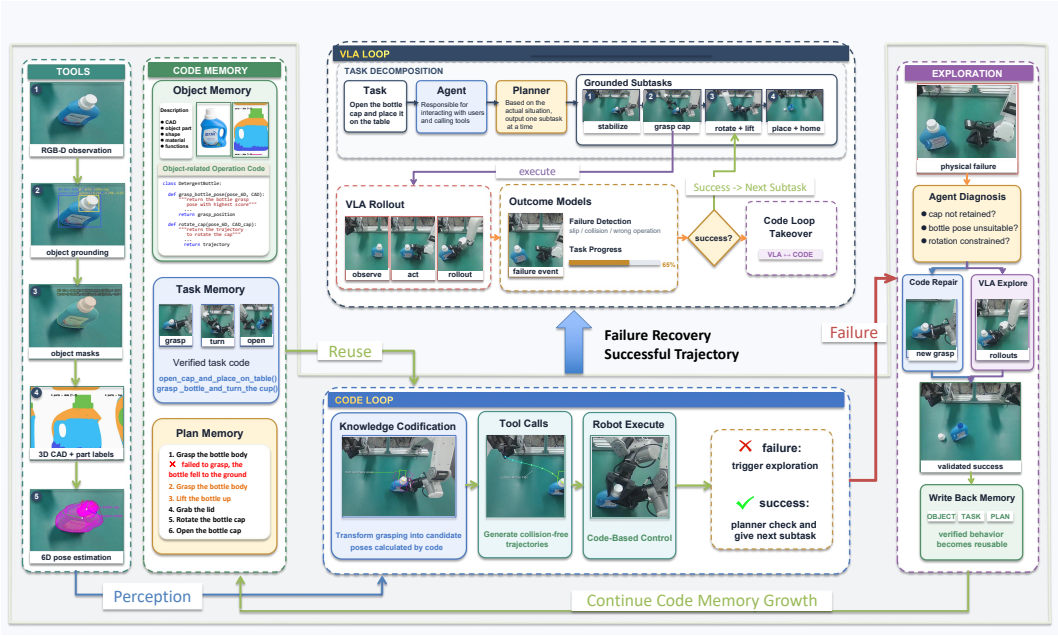


Figure 1: Overview of EvoRAM. The VLA and code loops collaborate with diverse tools to explore and execute tasks. Successful experience is consolidated into executable code memory and reused to strengthen subsequent exploration, forming an iterative self-evolution loop.

2 METHOD

In this section, we introduce EVORAM, an accumulative embodied agent framework that combines the complementary strengths of VLA policies and code-based agents. As illustrated in Fig. 1, we use the task of opening a bottle and placing its cap on the table as a running example to demonstrate the complete workflow of EVORAM. The framework operates through two interconnected loops: a **VLA loop** for efficient task execution and a **code loop** for deliberate reasoning, failure recovery, and experience accumulation.

In the inner loop, the VLA executes planner-proposed subtasks, with a verifier continuously tracking progress and detecting failures. When the VLA encounters situations beyond its capabilities, the outer loop invokes the code agent to reason over failures, use available tools, and explore alternative solutions. Diverse VLA behaviors can further assist code-based exploration, and successful executable solutions are consolidated into persistent executable memory for future reuse, allowing accumulated experience to strengthen subsequent exploration.

2.1 VLA LOOP

The VLA loop serves as the primary execution pathway of EVORAM, enabling efficient completion of tasks that fall within the existing capabilities of the VLA. As illustrated in Fig. 1, when opening a bottle, the VLA can successfully grasp the bottle but fails to open its cap. The VLA loop consists of three key components: a planner that progressively decomposes the user's instruction into executable subtasks, a VLA policy that interacts with the physical environment to execute each subtask, and a verifier that continuously monitors the execution progress and determines whether the current subtask is completed, failed, or still in progress.

Given a user instruction, EVORAM first acts as the interaction interface, maintaining the current task context while accommodating user interruptions and task updates. Based on the observation and execution history, the planner generates the next subtask for the VLA rather than producing a complete plan in advance. The VLA then executes the subtask through closed-loop interaction with the environment. Compared with explicit code-based control, the VLA provides faster and more flexible execution. Moreover, its stochastic policy naturally produces diverse interaction behaviors, providing an alternative exploration mechanism when deterministic code refinement becomes ineffective. Successful VLA trajectories can subsequently guide the code loop to acquire new solutions.

During execution, the verifier continuously evaluates the observed state. Once the current subtask succeeds or fails, the resulting state and verification feedback are returned to the planner, which updates its decision and generates the next subtask. This iterative process continues until the user's task is completed or the execution requires capabilities beyond those available in the VLA loop, in which case EVORAM transitions to the code loop described next.

2.2 CODE LOOP

When the VLA fails to complete a subtask, EVORAM switches to the code loop to solve problems beyond the VLA's current capabilities. Leveraging the reasoning and generalization capabilities of code agents, the code loop invokes diverse tools and writes executable programs to solve the task. When facing unseen problems without relevant experience, it continuously explores and verifies candidate solutions until a successful strategy is discovered, enabling self-evolution beyond its existing capabilities. Successful solutions are further consolidated into reusable code memory, allowing accumulated experience to facilitate future exploration and generalization. The resulting programs can also provide reliable failure recovery and corrective trajectories for failed VLA executions.

2.2.1 TOOLS

The code loop is equipped with perception and manipulation tools that bridge high-level reasoning and executable robot behaviors. Given RGB-D observations, the agent first identifies task-relevant objects and constructs object-centric representations. Specifically, the agent obtains object regions through vision-language guided perception and segmentation models (Team et al., 2025; Carion et al., 2026), reconstructs object-level CAD models (Xiang et al., 2025), and estimates accurate 6D poses using pose estimation methods (Wen et al., 2024). For objects with complex structures,

part-level decomposition (Liu et al., 2025) is further applied to enable fine-grained interaction with functional components. As illustrated in Fig. 1, through this perception pipeline, the agent obtains accurate poses of both the bottle and its cap for subsequent manipulation.

Based on the perception results, instead of relying on grasp prediction models such as AnyGrasp (Fang et al., 2023), the agent generates candidate interaction poses according to the task requirements. This design encodes object-specific manipulation knowledge in code, allowing successful interaction strategies to be persistently accumulated, reused, and adapted across tasks. The candidate poses are then evaluated with motion planning tools (Sundaralingam et al., 2023) to select feasible and collision-free motions. As illustrated in Fig. 1, through code-based motion control, the robot precisely grasps the bottle cap and moves to an appropriate pose for subsequent cap opening. The generated programs interact with robots through robot-agnostic manipulation interfaces, enabling the same code-level behaviors to be deployed across different robotic platforms.

In real-world environments, tools may produce imperfect results. Therefore, the code loop continuously verifies intermediate outputs through physical observations and refines the execution procedure when inconsistencies are detected. This verification-driven tool usage enables the agent to robustly explore and acquire executable solutions. Details on the tools are provided in Appendix B.

2.2.2 EXPLORATION

Building upon tools described above, the code loop can continuously explore solutions to tasks beyond its existing capabilities through iterative trial and verification. As illustrated in Fig. 1, when a task fails, the planner triggers the code loop to diagnose the failure based on observations collected at critical execution stages. Since robot behaviors in the code loop are expressed as explicit programs, EVORAM can instrument the generated code to capture intermediate observations, making each execution step traceable and facilitating fine-grained failure diagnosis.

Based on the diagnosed failure, EVORAM explores solutions through two complementary strategies. When the failure can be explicitly attributed, the agent directly revises code and re-executes it for verification. When repeated code refinement remains insufficient, EVORAM invokes the VLA for further exploration. The stochastic behaviors of the VLA provide diverse interaction trajectories, from which occasional successful executions can be identified and recorded. The code agent then uses these successful trajectories as guidance to revise its program, converting stochastic VLA successes into deterministic and reproducible code. The resulting program is finally executed and verified in the physical environment. Through this iterative process of diagnosis, exploration, and verification, EVORAM can progressively acquire executable solutions for previously unsolved tasks.

2.2.3 MEMORY

To preserve the experience acquired through exploration, we introduce a persistent code-based memory, as illustrated in Fig. 1, that stores verified experience primarily as executable code rather than natural language summaries. This representation preserves the complete execution procedure, allowing previously encountered problems to directly reuse verified solutions and related problems to adapt existing code for more efficient exploration. We organize the accumulated experience into three types of memory, which we introduce below.

Object Memory stores reusable experience associated with individual objects and consists of three components: an object description, a CAD representation, and an executable code class. The object description summarizes visual and physical characteristics, such as shape and material, and is primarily used to retrieve objects with relevant prior experience. The CAD representation records both the complete object geometry and task-relevant parts identified through part decomposition, such as the cap and body of a bottle, providing geometric references for code execution. Similar object instances can directly reuse existing representations and code, while objects with substantially different geometries are reconstructed and stored as new object types. Finally, the executable code class consolidates verified manipulation functions involving the object across different tasks, such as generating suitable interaction poses for grasping a bottle cap. More details are provided in Appendix C.

Task Memory stores the complete verified execution code for subtasks generated by the planner. Each entry is indexed by its subtask description, allowing the agent to retrieve relevant solutions for

reference or directly execute the code when encountering the same subtask. For example, a verified program for rotating a bottle cap can be repeatedly invoked when multiple rotations are required to fully open the bottle.

Plan Memory records planning failures and their corresponding corrections, primarily in natural language, to prevent the planner from repeating previously observed decision errors. For example, if a failed grasp knocks over a bottle, Plan Memory can instruct the planner to first restore the bottle to an upright state before continuing the opening procedure, rather than directly operating on the fallen bottle.

2.2.4 FAILURE RECOVERY

Beyond taking over subtasks that the VLA cannot complete, the code loop also supports fine-grained recovery from intermediate execution failures. When the verifier detects a failure during VLA execution, EVORAM analyzes the resulting physical state and generates executable code to continue from the failure state rather than restarting the task from scratch. As shown by the "failure event" in the VLA loop of Fig. 1, repeated VLA grasping attempts may fail and cause the bottle to slip away from its original position. The code loop can then re-localize the bottle and generate a corrective grasping trajectory from its current state. Such recoveries naturally produce verified trajectories connecting failed VLA states to successful outcomes, providing a potential source of dense supervision for future VLA improvement. For previously explored tasks, EVORAM further supports a *Quick Mode* that directly retrieves verified code from memory for efficient execution and failure recovery, details are provided in Appendix D.

3 EXPERIMENTS

We conduct extensive experiments in both simulated and real-world environments to evaluate three key capabilities of EVORAM. **(1) Self-Evolution:** whether exploration and memory accumulation can form a mutually reinforcing loop, where successful exploration expands executable memory and accumulated experience in turn facilitates subsequent exploration; **(2) Code–VLA Collaboration:** whether code and VLA can complement each other during exploration and execution, with VLA providing diverse behaviors to broaden code-based exploration and code providing reliable recovery from VLA failures; and **(3) Continuous Evolution:** whether EVORAM can autonomously maintain the physical capabilities required for sustained self-evolution under real-world changes such as calibration drift. We evaluate these capabilities on the LIBERO benchmark and a set of challenging real-world robotic tasks on our desktop dual-arm robotic platform. Detailed hardware and experimental configurations are provided in Appendix E.

3.1 SELF-EVOLUTION



Figure 2: Self-evolution through executable memory. Left: accumulated memory progressively facilitates task solving on LIBERO. Right: acquired experience facilitates exploration of a novel bottle-opening task.

We investigate whether EVORAM can continuously acquire new capabilities through a mutually reinforcing process between exploration and memory accumulation. We first study this process at

scale on LIBERO (Liu et al., 2023), where EvoRAM starts without prior experience and progressively acquires, reuses, and refines executable memory while solving a large collection of tasks. We then move to the real world and further examine whether the acquired experience can generalize beyond previously encountered objects to facilitate the exploration of novel ones.

Progressive Self-Evolution on LIBERO. We initialize EvoRAM with an empty memory and task it with solving all 130 LIBERO tasks using only the code loop to isolate the self-evolution driven by executable memory accumulation. To closely match the real-world setting, the agent only observes RGB-D images from the third-person and wrist cameras, while privileged simulator states and task predicates are hidden from the agent and used only for evaluation. The agent autonomously determines whether each task is completed and may temporarily skip tasks that it cannot solve, revisiting them after acquiring additional experience from other tasks.

As shown in Fig. 2, the evolution process exhibits three stages. In the first round, EvoRAM completes 80 tasks while accumulating 33 object memories containing 3,816 lines of executable code, establishing reusable experience for most objects in the benchmark. In the second round, equipped with these memories, EvoRAM revisits previously unsolved tasks and completes 126 tasks, while the number of object memories remains unchanged and the executable code increases by only about 200 lines. This indicates that accumulated object experience can be reused or adapted to solve many previously inaccessible tasks. In the final round, EvoRAM focuses on the remaining four tasks. Although all involved objects have already been encountered and stored in memory, previously verified programs fail under these new task configurations, requiring further exploration and adaptation. This process further refines the existing memories, increasing the executable code from 4,036 to 5,266 lines. After the complete exploration process, EvoRAM judges all tasks as solved, while 121/130 tasks (93.1%) are actually successful. A detailed case study is provided in Appendix F.

These results demonstrate how self-evolution and memory accumulation reinforce each other: exploration continuously acquires and refines executable experience, while the accumulated experience enables the agent to revisit and solve problems that were previously beyond its capabilities. We next ask whether such experience can go beyond reuse on previously encountered objects and facilitate exploration of novel objects.

Generalizing Experience to Novel Objects. We investigate this question through a real-world bottle-opening task. Starting without any relevant prior experience, EvoRAM requires approximately two hours of autonomous trial-and-error exploration to successfully open a detergent bottle and consolidate the verified solution into memory. We then replace it with a beverage bottle with substantially different geometry. Rather than exploring again from scratch, EvoRAM retrieves the previously acquired experience, adapts the corresponding code to the new geometry, and successfully opens the new bottle after approximately 40 minutes of exploration. This result shows that accumulated executable experience can not only be reused for previously encountered objects, but also provide a stronger starting point for self-evolution on novel but related objects. Detailed experimental procedures are provided in Appendix G.

Finding 1. EvoRAM exhibits continuous self-evolution: exploration expands memory, while accumulated memory strengthens subsequent exploration.

3.2 Code–VLA Collaboration

We next investigate how the code and VLA loops complement each other in real-world manipulation. The two loops exhibit distinct strengths and failure modes: the VLA provides flexible and reactive behaviors but may struggle with precise execution, whereas the code loop offers deterministic and reproducible control but can be limited when exploring complex interactions. We therefore evaluate their collaboration in both directions: **Code for VLA**, where verified code provides reliable recovery from VLA failures, and **VLA for Code**, where diverse VLA behaviors broaden the exploration space and help the code loop acquire new executable solutions.

3.2.1 Code for VLA

We first investigate whether the code loop can recover from tasks beyond the capabilities of the VLA loop. We conduct this experiment using EvoRAM's Quick Mode and consider a precision

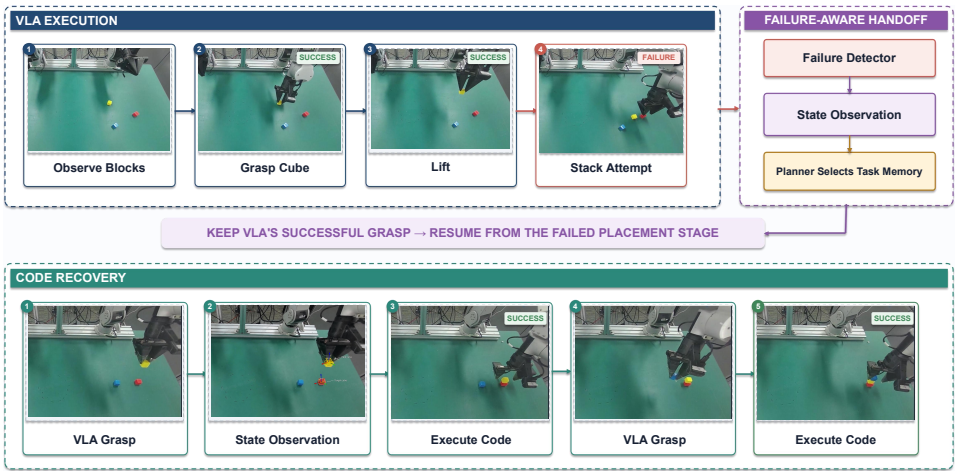


Figure 3: Code-based failure recovery in cube stacking. When placement exceeds the capability of the VLA loop, the code loop takes over from the failed state and completes the stacking task.

stacking task using 1.8 cm cubes. While the VLA loop can reliably grasp the cubes, it fails to precisely place them for stacking despite repeated attempts. As illustrated in Fig. 3, once the verifier identifies the subtask as failed, the code loop takes over from the resulting physical state, leverages its perception tools to accurately localize the cubes, and executes a precise corrective placement trajectory. Across 20 real-world trials, code-based takeover successfully completed two-layer stacking in 16 trials (80%), among which 5 trials further achieved three-layer stacking (25% overall). These results demonstrate that the code loop can extend task execution beyond the capability boundary of the VLA loop. Moreover, the recovered executions naturally provide verified corrective trajectories, which can serve as a potential source of dense supervision for future VLA improvement.

3.2.2 VLA FOR CODE

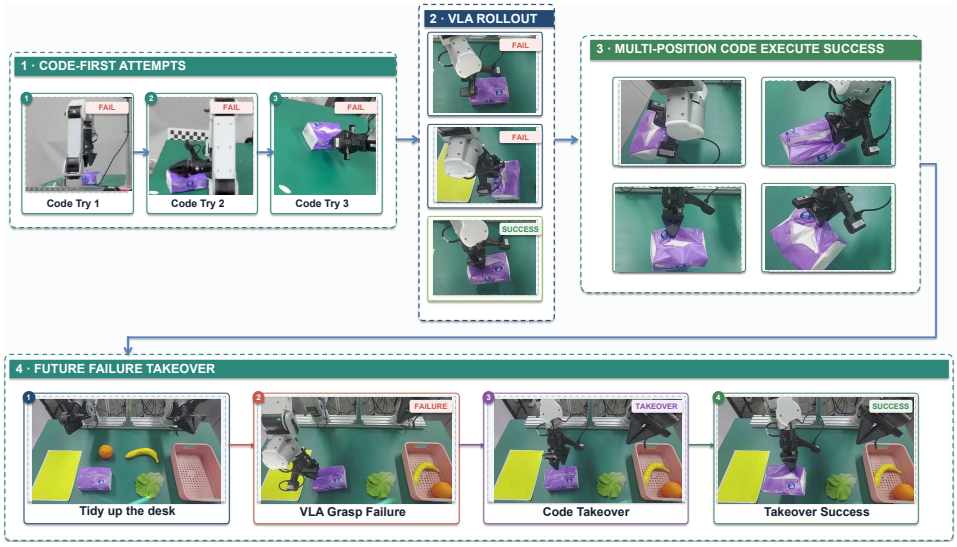


Figure 4: VLA-assisted exploration for tissue-pack grasping. VLA behaviors guide the code loop beyond failed exploration, enabling successful grasps to be consolidated into executable memory. The acquired memory subsequently recovers VLA grasping failures during table organization.

We next investigate whether the VLA loop can facilitate code-based exploration for interactions that are difficult to explicitly model. As illustrated in Fig. 4, we consider grasping a deformable

tissue pack, where repeated code-based exploration fails to discover a stable grasp. EvoRAM then invokes the VLA loop for stochastic exploration, from which a successful trajectory is identified by the verifier and used to guide the code loop in acquiring a reproducible solution. The resulting code successfully grasps the tissue pack under 8/10 different orientations. We further evaluate the acquired experience in a long-horizon sorting task: among five trials, the VLA loop succeeds directly in four, while in the remaining trial, the newly acquired code memory successfully recovers the failed tissue grasp and completes the task. These results show that VLA exploration can facilitate the acquisition of executable experience, which can in turn provide reliable recovery for subsequent VLA failures.

Finding 2. The complementary capabilities of code and VLA strengthen self-evolution by expanding the exploration space and accelerating the accumulation of executable memory.

3.3 Continuous Evolution

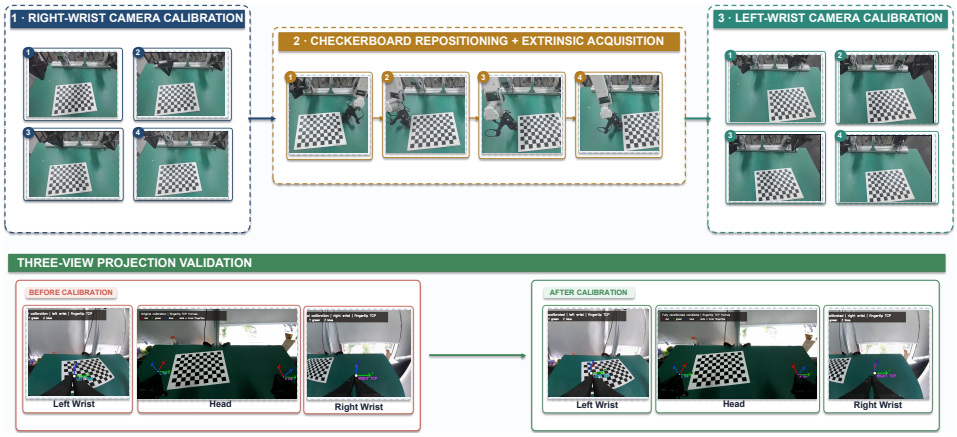


Figure 5: Autonomous recalibration for continuous self-evolution. EvoRAM manipulates the calibration board, collects multi-view observations, and restores degraded camera–robot calibration using accumulated experience.

Sustained self-evolution in the physical world requires maintaining accurate camera–robot calibration. However, calibration may gradually degrade during long-term operation, and maintaining calibration remains an important challenge for autonomous lifelong robotic systems (Lu et al., 2026; Bahadir et al., 2024). As shown on the left of the *Three-View Projection Validation* in Fig. 5, after completing our real-world experiments, the right gripper exhibits a noticeable projection error in the head camera. While such drift already affects precise code-based manipulation, manipulating the large calibration board requires relatively low positioning accuracy and therefore remains feasible under the degraded calibration. We thus investigate whether EvoRAM can autonomously recalibrate itself under such conditions.

Autonomous recalibration is a challenging long-horizon task involving board manipulation, multi-view acquisition, and both intrinsic and extrinsic calibration. Informative viewpoints must be explicitly computed from the current board pose, camera geometry, collected views, and robot reachability, making systematic acquisition difficult through reactive VLA execution alone. We first perform one human-in-the-loop exploration run, where the human only identifies incorrect subtasks, to establish the calibration procedure and consolidate the verified experience into executable memory. We then evaluate EvoRAM without human intervention. Leveraging the acquired memory, EvoRAM autonomously manipulates the board, computes and collects informative viewpoints, performs calibration, and verifies the resulting parameters in approximately one hour. As shown on the right of Fig. 5, the projection alignment is substantially restored after recalibration. Complete execution and calibration details are provided in Appendix H.

Finding 3. EvoRAM can autonomously recalibrate itself using accumulated experience, enabling sustained self-evolution under calibration degradation.

4 Related Work

Self-Evolving Embodied Agents. LLM-based robotic agents can decompose tasks, invoke tools, and generate executable programs for embodied control (Wang et al., 2023; Liang et al., 2023; Singh et al., 2023; Huang et al., 2023b). Recent work has begun moving beyond one-shot task solving toward continual capability acquisition through interaction. ETA (Chen et al., 2026) iteratively plans, executes, and adapts from environmental feedback, ENPIRE (Xiao et al., 2026) studies autonomous policy improvement through repeated physical experimentation, and ASPIRE (Lu et al., 2026) explores and refines robot programs while accumulating reusable skills. PhyAgentOS (Liu et al., 2026) further provides persistent infrastructure for long-term embodied agents, while Harness VLA (Zhang et al., 2026) combines coding agents with VLA control for reliable long-horizon execution. However, how exploration and accumulated experience can continuously reinforce each other to drive persistent capability growth remains underexplored. EvoRAM addresses this gap by forming a self-evolution loop in which exploration expands executable memory and accumulated memory strengthens subsequent exploration.

Memory for Embodied Agents. Memory has been widely explored as a mechanism for retaining temporal context and reusing past experience. Retrieval-Augmented Embodied Agents (Zhu et al., 2024) retrieve relevant policies from an external memory bank, while MemoryVLA (Shi et al., 2026), ReMem-VLA (Li et al., 2026), and MemER (Sridhar et al., 2025) incorporate perceptual, recurrent, or retrieved experience into VLA policies for long-horizon manipulation. At the agent level, EmbodiedCoder (Lin et al., 2025) reuses previously verified programs, while ASPIRE (Lu et al., 2026) continually expands a reusable skill library. Analytic Concept-Centric Memory (Sun et al., 2026) further organizes manipulation experience around structured objects, states, transitions, and executable skills. Unlike memory mechanisms primarily designed for contextual retrieval or skill reuse, EvoRAM organizes verified experience into persistent object, task, and planning memories represented predominantly as executable code. More importantly, memory is integrated into the self-evolution process: exploration continuously expands and refines executable memory, while accumulated experience provides stronger starting points for subsequent exploration.

5 Limitations and Future Work

Our current framework has four main limitations. (1) Real-world experiments are primarily conducted on a desktop robotic platform, where environmental setup still requires human assistance. Future work will extend EvoRAM to mobile and multi-robot systems toward persistent autonomous exploration in the physical world. (2) Although Quick Mode enables efficient reuse of verified experience, exploring complex unseen tasks can still take hours. Improving exploration efficiency through more effective tool orchestration remains an important direction. (3) The code loop remains limited in interactions that are difficult to explicitly model, such as complex manipulation of deformable objects or tightly cluttered small objects, where the VLA loop often plays a more important role. (4) While the code loop provides failure recovery and successful corrective trajectories for the VLA, these trajectories are not used to update the VLA in this work. Future work will leverage accumulated recovery trajectories to improve end-to-end VLA policies, whose enhanced capabilities can in turn facilitate exploration and further strengthen the self-evolution loop.

6 Conclusion

We presented EvoRAM, a self-evolving embodied agent that continuously transforms successful exploration into persistent executable experience. By combining code-based exploration with VLA execution, EvoRAM can explore beyond existing capabilities, physically verify discovered solutions, and consolidate them into structured executable memory. Accumulated experience can then be reused, adapted, and refined to facilitate subsequent exploration, forming a self-evolution loop in which exploration expands memory and memory strengthens exploration.

AI USE STATEMENT

In this work, we used generative AI tools to improve the readability of author-written text, assist with literature search and retrieval, and create or edit software code. We reviewed all AI-assisted work. Retrieved references were manually verified against their original sources, and AI-assisted code was reviewed and tested for correctness. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies self-evolving embodied agents that autonomously explore and execute generated programs on robotic systems. Physical deployment requires appropriate safety constraints, particularly as newly generated behaviors may lead to unintended robot motions or interactions with the environment. In our experiments, physical execution is constrained by predefined robot interfaces, collision-aware motion planning, workspace constraints, and emergency stop mechanisms. Broader deployment in less controlled environments would require additional safety verification and safeguards appropriate to the operating environment and interactions with people and objects.

REPRODUCIBILITY STATEMENT

Appendices A–D detail the agent implementation, tools, memory, and execution modes; Appendix E specifies the hardware and model configurations; Appendices F–H provide complete experimental procedures and analyses; and Appendix I documents the prompts and model inputs. The supplementary material provides our implementation and the executable code memory accumulated throughout the experiments. We additionally provide demonstration videos of the real-world experiments.

REFERENCES

Ozan Bahadir, Jan Paul Siebert, and Gerardo Aragon-Camarasa. Continual learning approaches to hand–eye calibration in robots. *Mach. Vision Appl.*, 35(4), July 2024. ISSN 0932-8092. doi: 10.1007/s00138-024-01572-w. URL https://doi.org/10.1007/s00138-024-01572-w.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control, 2026. URL https://arxiv.org/abs/2410.24164.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. RT-2: Vision-language-action models transfer web knowledge to robotic control. In *Proceedings of the 7th Conference on Robot Learning*, 2023.

Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, Jie Lei, Tengyu Ma, Baishan Guo, Arpit Kalla, Markus Marks, Joseph Greer, Meng Wang, Peize Sun, Roman Rädle, Triantafyllos Afouras, Effrosyni Mavroudi, Katherine Xu, Tsung-Han Wu, Yu Zhou, Liliane Momeni, Rishi Hazra, Shuangrui Ding, Sagar Vaze, Francois Porcher, Feng Li, Siyuan Li, Aishwarya Kamath, Ho Kei Cheng, Piotr Dollár, Nikhila Ravi, Kate Saenko, Pengchuan Zhang, and Christoph Feichtenhofer. Sam 3: Segment anything with concepts, 2026. URL https://arxiv.org/abs/2511.16719.

Yitong Chen, Zezheng Huai, Sixian Li, Yubang Wang, Haozhe Zhang, Yifei Zhang, Hechang Chen, Jingjing Gong, Yu-Gang Jiang, and Xipeng Qiu. Eta: A new agentic paradigm for embodied tasks, 2026. URL https://arxiv.org/abs/2608.03924.

Hao-Shu Fang, Chenxi Wang, Hongjie Fang, Minghao Gou, Jirong Liu, Hengxu Yan, Wenhai Liu, Yichen Xie, and Cewu Lu. Anygrasp: Robust and efficient grasp perception in spatial and temporal domains. *IEEE Transactions on Robotics (T-RO)*, 2023.

Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model, 2023a. URL https://arxiv.org/abs/2305.11176.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *Conference on Robot Learning (CoRL)*, 2023b.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model, 2024. URL https://arxiv.org/abs/2406.09246.

Hang Li, Fengyi Shen, Dong Chen, Liudi Yang, Xudong Wang, Jinkui Shi, Zhenshan Bing, Ziyuan Liu, and Alois Knoll. Remem-vla: Empowering vision-language-action model with memory via dual-level recurrent queries, 2026. URL https://arxiv.org/abs/2603.12942.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Zefu Lin, Rongxu Cui, Chen Hanning, Xiangyu Wang, Junjia Xu, Xiaojuan Jin, Chen Wenbo, Hui Zhou, Lue Fan, Wenling Li, and Zhaoxiang Zhang. Embodiedcoder: Parameterized embodied mobile manipulation via modern coding model, 2025. URL https://arxiv.org/abs/2510.06207.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 44776–44791. Curran Associates, Inc., 2023. doi: 10.52202/075280-1939. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/8c3c666820ea055a77726d66fc7d447f-Paper-Datasets_and_Benchmarks.pdf.

Minghua Liu, Mikaela Angelina Uy, Donglai Xiang, Hao Su, Sanja Fidler, Nicholas Sharp, and Jun Gao. Partfield: Learning 3d feature fields for part segmentation and beyond, 2025. URL https://arxiv.org/abs/2504.11451.

Yang Liu, Weixing Chen, Xinshuai Song, Tao Pu, Siwen Mo, Yongjie Bai, Zihao Chen, Qianran Sun, Liruo Zhong, Ying Shen, and Liang Lin. Phyagentos: A self-evolving operating system for embodied agents with decoupled cognitive planning and physical execution, 2026. URL https://arxiv.org/abs/2607.16636.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi "Jim" Fan, and Guanzhi Wang. Aspire: Agentic /skills discovery for robotics, 2026. URL https://arxiv.org/abs/2607.00272.

Abby O'Neill, Abdul Rehman, and Maddukuri et al. Open x-embodiment: Robotic learning datasets and rt-x models, 2025. URL https://arxiv.org/abs/2310.08864.

Hao Shi, Bin Xie, Yingfei Liu, Lin Sun, Fengrong Liu, Tiancai Wang, Erjin Zhou, Haoqiang Fan, Xiangyu Zhang, and Gao Huang. Memoryvla: Perceptual-cognitive memory in vision-language-action models for robotic manipulation, 2026. URL https://arxiv.org/abs/2508.19236.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Ajay Sridhar, Jennifer Pan, Satvik Sharma, and Chelsea Finn. Memer: Scaling up memory for robot control via experience retrieval, 2025. URL https://arxiv.org/abs/2510.20328.

Mingyang Sun, Xiujian Liang, Jiude Wei, Qichen He, Donglin Wang, Cewu Lu, and Jianhua Sun. Analytic concept-centric memory for agentic embodied manipulation, 2026. URL https://arxiv.org/abs/2606.29774.

Balakumar Sundaralingam, Siva Kumar Sastry Hari, Adam Fishman, Caelan Garrett, Karl Van Wyk, Valts Blukis, Alexander Millane, Helen Oleynikova, Ankur Handa, Fabio Ramos, Nathan Ratliff, and Dieter Fox. curobo: Parallelized collision-free minimum-jerk robot motion generation, 2023. URL https://arxiv.org/abs/2310.17274.

Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, Steven Bohez, Konstantinos Bousmalis, Anthony Brohan, Thomas Buschmann, Arunkumar Byravan, Serkan Cabi, Ken Caluwaerts, Federico Casarini, Oscar Chang, Jose Enrique Chen, Xi Chen, Hao-Tien Lewis Chiang, Krzysztof Choromanski, David D'Ambrosio, Sudeep Dasari, Todor Davchev, Coline Devin, Norman Di Palo, Tianli Ding, Adil Dostmohamed, Danny Driess, Yilun Du, Debidatta Dwibedi, Michael Elabd, Claudio Fantacci, Cody Fong, Erik Frey, Chuyuan Fu, Marissa Giustina, Keerthana Gopalakrishnan, Laura Graesser, Leonard Hasenclever, Nicolas Heess, Brandon Hernaez, Alexander Herzog, R. Alex Hofer, Jan Humplik, Atil Iscen, Mithun George Jacob, Deepali Jain, Ryan Julian, Dmitry Kalashnikov, M. Emre Karagozler, Stefani Karp, Chase Kew, Jerad Kirkland, Sean Kirmani, Yuheng Kuang, Thomas Lampe, Antoine Laurens, Isabel Leal, Alex X. Lee, Tsang-Wei Edward Lee, Jacky Liang, Yixin Lin, Sharath Maddineni, Anirudha Majumdar, Assaf Hurwitz Michaely, Robert Moreno, Michael Neunert, Francesco Nori, Carolina Parada, Emilio Parisotto, Peter Pastor, Acorn Pooley, Kanishka Rao, Krista Reymann, Dorsa Sadigh, Stefano Saliceti, Pannag Sanketi, Pierre Sermanet, Dhruv Shah, Mohit Sharma, Kathryn Shea, Charles Shu, Vikas Sindhwani, Sumeet Singh, Radu Soricut, Jost Tobias Springenberg, Rachel Sterneck, Razvan Surdulescu, Jie Tan, Jonathan Tompson, Vincent Vanhoucke, Jake Varley, Grace Vesom, Giulia Vezzani, Oriol Vinyals, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Fei Xia, Ted Xiao, Annie Xie, Jinyu Xie, Peng Xu, Sichun Xu, Ying Xu, Zhuo Xu, Yuxiang Yang, Rui Yao, Sergey Yaroshenko, Wenhao Yu, Wentao Yuan, Jingwei Zhang, Tingnan Zhang, Allan Zhou, and Yuxiang Zhou. Gemini robotics: Bringing ai into the physical world, 2025. URL https://arxiv.org/abs/2503.20020.

Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, You Liang Tan, Lawrence Yunliang Chen, Pannag Sanketi, Quan Vuong, Ted Xiao, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An open-source generalist robot policy, 2024. URL https://arxiv.org/abs/2405.12213.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities, 2023. URL https://arxiv.org/abs/2306.17582.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Bowen Wen, Wei Yang, Jan Kautz, and Stan Birchfield. Foundationpose: Unified 6d pose estimation and tracking of novel objects, 2024. URL https://arxiv.org/abs/2312.08344.

Jianfeng Xiang, Xiaoxue Chen, Sicheng Xu, Ruicheng Wang, Zelong Lv, Yu Deng, Hongyuan Zhu, Yue Dong, Hao Zhao, Nicholas Jing Yuan, and Jiaolong Yang. Native and compact structured latents for 3d generation, 2025. URL https://arxiv.org/abs/2512.14692.

Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian "Max" Fu, Haoru Xue, Jalen Lu, Yi Yang, Cunxi Dai, Zi Wang, Jimmy Wu, Guanzhi Wang, S. Shankar Sastry, Ken Goldberg, Linxi "Jim"

Fan, Yuke Zhu, and Guanya Shi. Enpire: Agentic robot policy self-improvement in the real world, 2026. URL https://arxiv.org/abs/2606.19980.

Andy Zhai, Brae Liu, Bruno Fang, Chalse Cai, Ellie Ma, Ethan Yin, Hao Wang, Hugo Zhou, James Wang, Lights Shi, Lucy Liang, Make Wang, Qian Wang, Roy Gan, Ryan Yu, Shalfun Li, Starrick Liu, Sylas Chen, Vincent Chen, and Zach Xu. Igniting vlms toward the embodied space, 2025. URL https://arxiv.org/abs/2509.11766.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, Zhengru Fang, Yi Nie, Changxu Wei, Yu Wang, Wenbo Ding, and Chao Yu. Harness vla: Steering frozen vlas into reliable manipulation primitives via memory-guided agents, 2026. URL https://arxiv.org/abs/2607.08448.

Yichen Zhu, Zhicai Ou, Xiaofeng Mou, and Jian Tang. Retrieval-augmented embodied agents, 2024. URL https://arxiv.org/abs/2404.11699.

A AGENT DETAILS

A.1 RELATION TO EXISTING EMBODIED AGENTS

Table 1 compares the mechanisms most relevant to our self-evolution setting. The comparison concerns reported system designs, rather than success rates obtained under different robots, observation interfaces, and evaluation budgets. In particular, persistent context, reusable textual guidance, and directly executable memory are related but distinct representations. We distinguish them to clarify what is accumulated and how it is reused.

Comparison criteria. *Open-ended exploration* refers to revising or discovering executable behaviors through interaction, beyond selecting from a fixed action menu. *Persistent executable memory* stores verified experience in directly executable forms that can be invoked, reused, or adapted in future interactions, rather than only as textual or abstract knowledge. *Continual memory guidance* requires accumulated experience to remain reusable and refinable for guiding subsequent exploration as memory grows, rather than only supporting isolated task-specific reuse. *VLA collaboration* combines policy execution with agentic planning or code-based control; it does not require every method to implement our particular two-way loop. *Autonomous physical maintenance* denotes demonstrated physical procedures that restore the system's operating capability, such as manipulating a calibration target and obtaining new camera–robot parameters, rather than software restart or memory management alone.

Table 1: Qualitative comparison of mechanisms relevant to persistent self-evolution. **Yes**: explicitly supported in the reported system. **Partial**: a related mechanism is present, but its representation, scope, or demonstrated use differs from the definition below. **–**: not demonstrated in the cited work, rather than an assertion that the architecture could never support it. Executable memory denotes persistent, reusable implementations, not merely the ability to generate code during one episode.

Method	Open-ended exploration	Executable memory	Continual memory guidance	VLA collaboration	Physical maintenance
ETA (Chen et al., 2026)	Yes	–	–	–	–
ENPIRE (Xiao et al., 2026)	Yes	–	–	Yes	–
ASPIRE (Lu et al., 2026)	Yes	Yes	Partial	–	–
Harness VLA (Zhang et al., 2026)	Partial	–	Partial	Yes	–
PhyAgentOS (Liu et al., 2026)	Partial	–	Partial	Yes	–
EmbodiedCoder (Lin et al., 2025)	Partial	Yes	Partial	–	–
EVORAM (ours)	Yes	Yes	Yes	Yes	Yes

Interpretation of partial and unsupported capabilities. ETA maintains an editable textual experience that guides subsequent tool use, but the stored experience itself does not directly reproduce acquired manipulation behaviors. Its reported system also does not establish continual memory-guided exploration or bidirectional code–VLA collaboration (Chen et al., 2026). ENPIRE autonomously improves executable robot policies through physical experimentation and transfers learned recipes across experiments, but does not maintain a structured executable memory that continually grows and guides exploration across a broad task collection (Xiao et al., 2026). ASPIRE accumulates validated executable skills and retrieves them to facilitate subsequent exploration. However, its memory is organized primarily around task-level skills, making experience difficult to disentangle and compose across tasks. The authors further acknowledge that, as the skill library grows, accumulated skills may become stale, overly specific, redundant, or misleading; we therefore classify continual memory guidance as partial (Lu et al., 2026).

Harness VLA retains execution traces, success rules, and failure models to guide the composition of analytic primitives and VLA behaviors, but does not continually synthesize and accumulate new executable skills (Zhang et al., 2026). PhyAgentOS consolidates verified outcomes primarily into persistent textual knowledge and corrective lessons, while executable skills are handled separately by its runtime; we therefore do not classify its accumulated experience as persistent executable memory. Although this experience can guide subsequent planning, its evaluation does not establish progressively stronger exploration as memory grows, resulting in partial continual memory guidance (Liu

et al., 2026). EmbodiedCoder caches verified executable programs for subsequent reuse, but does not center on continually refining a growing memory to support open-ended exploration (Lin et al., 2025).

Algorithm 1: Execution and self-evolution in EVORAM

Input: user goal g; Robot interface R; planner P; verifier V; tools $\mathcal{T}$; optional VLA policy π; exploration budget B.
Persistent state: Object, Task, and Plan Memory $\mathcal{M}$; deferred tasks $\mathcal{D}$.
Working state: observations o, execution history H, candidate program c.

while the current goal is active and resources are available **do**
  Handle stop or goal-update requests; stop the old execution before replanning.
  $o \leftarrow R.\mathrm{observe}()$; retrieve relevant Plan Memory and update H.
  $s \leftarrow P(g, o, H, \mathcal{M}_{\mathrm{plan}})$, one next subtask or completion.
  if P judges the parent goal complete from current evidence, **return**.
  if VLA-first execution is enabled **then**
    Run $\pi(s)$; use V to monitor fresh observations until success or failure.
    Record the outcome in H; **if** successful, **continue** at line 1.
    Stop VLA on confirmed failure; acquire the resulting scene.
  end if
  Retrieve relevant Object and Task Memory; read their implementations.
  $c \leftarrow$ reuse, adapt, or write a program for s from the current state.
  while s is unresolved and its exploration budget remains **do**
    Refresh perception; inspect object identities, geometry, masks, and poses.
    Generate task-conditioned pose candidates and evaluate motions with R.
    Execute the feasible program c; save traces and fresh outcome images.
    if the executed code is verified successful **then**
      Consolidate reusable object functions and the verified task program.
      Update applicable planning lessons and evidence links; **break**.
    end if
    Diagnose the failed stage using the trace and observed physical outcome.
    if code refinement has stalled and VLA exploration is enabled **then**
      Run targeted VLA attempts only for the unresolved interaction.
      Identify and record successful attempts using the verifier.
      Infer reusable contact regions and motion constraints from successes.
      $c \leftarrow$ a generalized program guided by this experience.
    else $c \leftarrow$ revise the program using the diagnosed failure.
    end if; return to line 14 to execute and verify c.
  end while
  Update H; if still unresolved, record the failure and defer or replan s.
end while; revisit tasks in $\mathcal{D}$ as relevant experience accumulates.

A.2 Complete Execution and Memory-Accumulation Procedure

EVORAM maintains the current user goal, execution history, active subtask, and three persistent memory stores. The planner receives fresh observations and the outcome of the preceding action, and proposes one next subtask. It does not commit the robot to a complete open-loop plan. A user update invalidates the remaining old plan: the current execution is stopped, the resulting scene is observed, and planning resumes with both the interaction history and the revised goal.

Algorithm 1 summarizes the complete process. The default pathway is VLA-first. Once a confirmed VLA failure enters the outer loop, code-based diagnosis and refinement are attempted first. VLA exploration is then available as a tool for the unresolved interaction, rather than requiring the robot to repeat already successful stages.

B Tool Details

The tool layer gives EVORAM explicit access to perception, motion planning, execution, and stored experience. Each registered tool has a name, a description, a typed argument schema, and a structured return value. The model receives the schemas alongside the conversation; detailed instructions

and existing implementations are read when needed. Tool calls, return values, errors, and saved artifacts form the execution trace used for diagnosis. Tables 2–4 list the 30 registered interfaces, grouped by function.

The corresponding task-facing prompts and structured model inputs are documented in Appendix I. Tool schemas are supplied separately from these natural-language instructions.

Table 2: Planning and perception tools. The perception stages preserve intermediate artifacts so that downstream actions need not treat an earlier model output as unquestionable ground truth.

Tool	Function and returned information
`planner_reset`	Initialize a planning context for a new parent task.
`planner_next_subtask`	Use current views and outcome history to request one grounded subtask or parent-task completion.
`perception_capture`	Capture aligned RGB-D and a world-frame point cloud through Robot; initialize the environment artifact.
`perception_detect_objects`	Run task-conditioned VLM detection and record object labels and image regions.
`perception_segment`	Segment detected objects with SAM 3; save masks and visible object point clouds.
`perception_build_cad`	Assign a memory CAD to each object, or generate a new CAD with TRELLIS.2.
`perception_pose`	Estimate or track object 6D poses using FoundationPose from RGB-D, masks, and CAD; optionally inspect pose–mask agreement for precision tasks.
`perception_process_environment`	Organize all acquired perception information and current TCP poses into a JSON file for agent inspection.

Table 3: Robot tools. Read-only feasibility evaluation and physical execution are separate operations. The Robot layer owns embodiment-specific control and coordinate conversion.

Tool	Function and returned information
`robot_status`	Read service health, robot description, available arms and cameras, joints, TCPs, and grippers.
`robot_observe`	Save fresh RGB and optional depth observations from selected cameras.
`robot_record_video`	Record a specified camera for a bounded duration.
`robot_evaluate_end_pose`	Evaluate a world-frame TCP target without moving the robot.
`robot_evaluate_grasp_candidates`	Evaluate candidate approach/contact poses and return motion feasibility and ranking information without execution.
`robot_move_tcp`	Plan and execute a collision-aware move to a world-frame TCP target.
`robot_execute_joint_trajectory`	Execute a previously prepared joint trajectory and report execution status.
`robot_execute_tcp_trajectory`	Execute a specified TCP trajectory through the selected robot backend.
`robot_set_gripper`	Set the selected gripper opening and report the operation result.
`robot_home`	Request homing and wait for the corresponding controller state.
`robot_stop`	Stop robot execution through the active backend.

Table 4: Code, memory, service, VLA tools. Permission to execute physical actions is controlled separately from read-only inspection.

Tool	Function and returned information
`read`	Read skill instructions, source code, metadata, and saved execution artifacts.
`write`	Create source code or other experiment files.
`edit`	Apply targeted modifications to existing files.
`bash`	Run generated programs and existing service clients; return process output and status.
`skill_list`	Discover available workflows and their instruction-file locations.
`model_service_list`	Inspect configured model services and interfaces.
`model_service_health`	Check whether the required model services are available.
`memory_search`	Return top-N Object, Task, and Plan Memory entries, with metadata and implementation locations.
`vla_run`	Start a bounded VLA execution with the specified instruction and optional outcome monitoring.
`vla_status`	Read VLA progress, verifier decisions, and associated trace locations.
`vla_stop`	Stop the active VLA execution.

C Object Memory Details

C.1 Representation and Retrieval

An Object Memory entry is identified by an object category and a geometry type. It consists of a searchable object description, metric CAD geometry, and an executable code class. The description records visual and physical characteristics, operated parts, supported functions, and relevant failure experience, enabling the agent to retrieve related objects from memory. The CAD representation provides the geometric reference for localization and manipulation, with optional part labels distinguishing functional regions. The code class stores verified object-specific manipulation functions, including contact preferences, approach constraints, stable support directions, and task-dependent interaction poses.

Taking the detergent bottle as an example, its code class contains functions for generating grasp poses, aligning with the cap, rotating the cap, testing whether the cap has been detached, and retreating after release. These object-level functions can be shared across different task programs, such as stabilizing the bottle, repeatedly rotating the cap, verifying cap removal, and placing the removed cap. When encountering a new bottle with substantially different geometry, the agent can reuse the functional knowledge from the existing Object Memory while reconstructing the CAD model and adapting the corresponding interaction poses and contact regions. When the same bottle appears at a different location or orientation, the stored functions can be directly instantiated using its newly estimated pose.

Retrieval is an index operation followed by model reasoning. The `memory_search` tool reads the Object, Task, and Plan indexes and returns the top-N matching entries; the default is five. Ranking combines normalized token overlap, object-key overlap, string similarity, and phrase matching over the query and indexed metadata. The agent then reads the selected implementations and decides whether to invoke, adapt, or reject them.

Memory boundaries. Object Memory stores reusable object-level operations; Task Memory stores verified compositions that implement subtasks; Plan Memory stores corrections to task ordering and interpretation. For example, keeping a stabilizing grasp below the cap belongs to object interaction knowledge, the entire stabilize–turn–test sequence belongs to a task program, and restoring a fallen bottle before opening it belongs to planning experience.

Table 5: Complete Object Memory inventory across physical and simulated tasks. The Objects column counts indexed object types.

Group	Objects	Object Memory types
Food and produce	4	Banana, orange, pear, lettuce leaf.
Bowls, dishes, and utensils	6	Bowl, spoon, Akita black bowl, white bowl, glazed-rim porcelain ramekin, plate.
Handled vessels	5	Porcelain mug, red coffee mug, white–yellow mug, moka pot, Chefmate frying pan.
Packages	9	Tissue pack, alphabet soup, tomato sauce, butter, chocolate pudding, cookies, cream cheese, milk, orange juice.
Bottles	8	Detergent bottle, mineral water bottle, sports drink bottle, BBQ sauce, ketchup, salad dressing, new salad dressing, wine bottle.
Books	2	Black book, yellow book.
Containers and supports	7	Square basket, rectangular paper, basket, desk caddy, wooden tray, wine rack, wooden two-layer shelf.
Articulated fixtures and appliances	4	White cabinet, wooden cabinet, microwave, flat stove.
Blocks and calibration targets	2	Cube block, calibration checkerboard.
Total	47	Complete accumulated library.

C.2 Accumulated Object Experience

Table 5 summarizes the complete Object Memory library accumulated after all physical-robot and simulation experiments. The library contains **47 indexed object types**, each associated with its own metadata, CAD representation, and executable implementation. Multiple scene instances can share the same type, while operated parts such as a bottle cap or drawer handle remain within their parent object's memory rather than being counted separately.

The resulting library spans grasping and placement, compliant-package contact, precise cube stacking, part-conditioned bottle manipulation, and calibration-target interaction. It also includes containers, supports, and articulated fixtures, whose reusable knowledge captures valid placement regions, support constraints, and opening or closing interactions in addition to object grasping.

Scope of the growth statistics. The complete library contains 33 object types accumulated from LIBERO and 14 additional types accumulated from physical-robot experiments, for a total of 47. The 33-object count in the LIBERO growth curve therefore refers only to the simulation subset, whereas Table 5 reports the complete memory accumulated after all experiments.

Growth within an existing object. Object Memory can grow without introducing a new object type. For example, a bottle that can already be grasped may later acquire cap-turning and separation-test functions, while a mug that can be placed on an open surface may require a new release strategy inside an appliance. Such experience enriches the executable code class and associated constraints of an existing memory. Object Memory therefore grows both by acquiring new object types and by expanding the executable capabilities associated with existing ones.

D Quick Mode

In addition to the agent-driven exploration process described above, we introduce a **Quick Mode** to efficiently and reliably execute previously explored tasks. As illustrated in Fig. 6, Quick Mode bypasses tool invocation, code generation, and iterative exploration by directly reusing verified experience stored in memory. Given a user instruction, the planner progressively generates subtasks that are first executed by the VLA. If the VLA fails, EvoRAM retrieves the corresponding verified program from Task Memory and directly executes it to recover the failed subtask. The planner then proceeds to the next subtask until the overall task is completed. By eliminating repeated exploration, Quick Mode enables efficient reuse of acquired capabilities while naturally collecting code-based recovery trajectories that can serve as supervision for future VLA improvement.

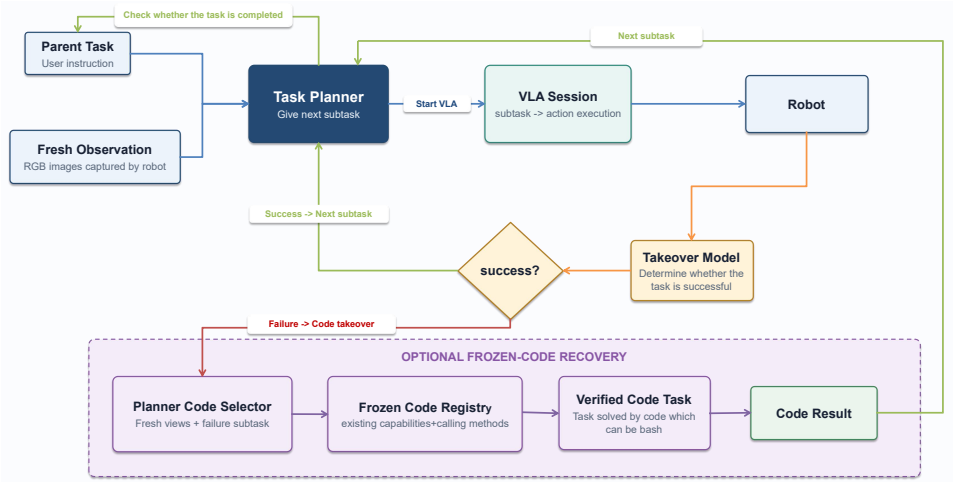
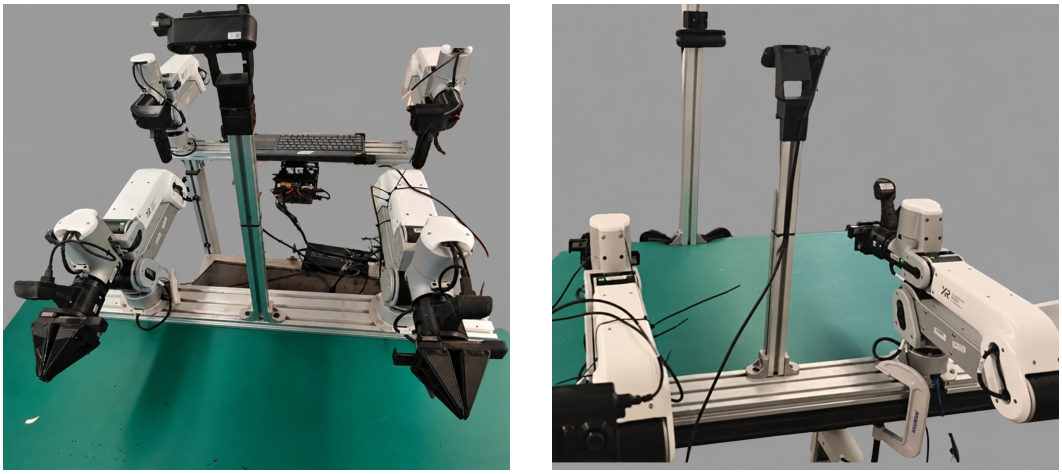


Figure 6: Quick Mode for efficient experience reuse. Verified code is directly retrieved from memory for task execution and VLA failure recovery without repeated exploration.

E Hardware and Experimental Configuration

E.1 Physical Platform and Observation Interfaces

Our physical experiments use a commercial desktop leader–follower dual-arm platform with parallel-jaw grippers. The manipulation workspace is observed by an Orbbec DCW2 front camera, two Union Image fisheye wrist cameras, and an Orbbec Gemini 336L rear RGB-D camera. Figure 7 shows the front and rear views of the platform, including the workspace, camera placement, and arm layout. Autonomous execution uses the robot control interface without teleoperating the leader arms.



(a) Front view. (b) Rear view.

Figure 7: Physical experimental platform. Front and rear photographs show the dual-arm workspace and the cameras used for visual reasoning and geometric manipulation.

For physical task planning, the planner receives labeled views from the left wrist, front, and right wrist cameras. Code-based geometric manipulation obtains aligned RGB-D observations through the Robot interface, typically using the rear camera for scene-level geometric perception. Perception and control operate in a shared world frame, while the robot backend handles kinematics and controller-specific details.

E.2 Models and Execution Modes

The code agent uses GPT-5.6 Sol with the Ultra reasoning setting. Gemini Robotics ER 2 is used for visual task reasoning and planning, alongside the perception models listed in Appendix B. Physical VLA execution uses Wall-X (Zhai et al., 2025) with a separate outcome-monitoring model that tracks successful, failed, and ongoing execution. The planner generates the next subtask, the VLA executes it, and the code agent performs tool-augmented exploration and failure recovery.

All model parameters remain fixed during exploration, with adaptation occurring through executable programs, geometric representations, and accumulated memory. Physical experiments use VLA-first execution with code-based exploration and recovery, while the LIBERO evaluation uses only the code loop. Quick Mode directly reuses verified task programs, whereas exploration mode can retrieve, adapt, and expand accumulated memory.

F LIBERO Case Analysis

The main paper presents the overall progression of self-evolution on LIBERO; here, we further examine the execution histories underlying this process. We conduct the entire LIBERO experiment using only the code loop, without VLA execution, allowing us to isolate the self-evolution process driven by code-based exploration and memory accumulation. As exploration proceeds, EvoRAM progressively accumulates executable experience and revisits previously unsolved tasks with the acquired memory, exhibiting the self-evolution behavior reported in the main paper. In this section, we provide detailed case analyses of three representative stages: (1) tasks successfully revisited after relevant memory becomes available, illustrating how accumulated experience facilitates subsequent exploration; (2) the final four difficult tasks, which require substantial refinement of existing memory despite involving previously encountered objects; and (3) the nine tasks judged successful by the agent but identified as failures.

F.1 Representative Cases of Memory-Assisted Revisit

A key behavior during LIBERO self-evolution is that previously deferred tasks can become solvable after relevant experience is accumulated elsewhere. On revisit, retrieved Object and Task Memory changes the starting point of exploration: known interactions can be directly reused, multiple acquired capabilities can be composed, or existing procedures can be adapted to new configurations. We illustrate these three forms of memory-assisted exploration through representative execution histories below.

Direct reuse of acquired interactions. The clearest cases occur when experience acquired from one task directly provides an interaction procedure required by another.

- *Open the bottom drawer.* This task initially remains unresolved. Through continued exploration, EvoRAM eventually acquires a reliable drawer-opening procedure based on short contact motions followed by re-observation and correction. The verified procedure is consolidated into memory and becomes reusable for subsequent drawer-related tasks.
- *Close the microwave.* Early attempts fail to reliably manipulate the microwave door. Continued exploration produces a verified door-closing procedure, which is retained in memory and later reused when microwave closing appears as part of a more complex task.

Composition of accumulated capabilities. For compound instructions, memory allows independently acquired object and fixture interactions to be composed, shifting exploration toward their coordination rather than rediscovering each component.

- *Turn on the stove and put the frying pan on it.* The compound instruction is initially incomplete. After EvoRAM acquires reusable frying-pan grasping, transport, and support-placement experience, the revisit combines these operations with stove interaction. Exploration can therefore focus on coordinating the two capabilities rather than rediscovering pan manipulation.
- *Put the chocolate pudding in the top drawer and close it.* The initial attempt does not complete both placement and drawer closing. On revisit, EvoRAM combines accumulated package-

manipulation experience with the previously acquired drawer-operation procedure, yielding a complete execution of the compound task.

- *Put the yellow-and-white mug in the microwave and close it.* Earlier mug manipulation does not complete the full instruction. The revisit composes mug-placement experience with the microwave-closing procedure acquired above. Further exploration is limited to coordinating confined placement with the transition to door closing rather than learning either capability anew.

Adaptation to new configurations. In other cases, memory provides only part of the solution. The agent retains successful interaction stages and concentrates new exploration on geometry, target relations, or constrained placement conditions that differ from previous experience.

- *Put the wine bottle in the bottom drawer.* Existing bottle-grasping, transport, and drawer-interaction experience provides the basic solution structure, while the constrained bottle-to-drawer placement requires further adaptation to the current geometry.
- *Put the frying pan on the cabinet shelf.* The agent already possesses a verified pan-grasping procedure when this task is revisited. It therefore retains the successful grasp and concentrates exploration on transporting and releasing the pan into a stable configuration on the shelf.
- *Pick up the middle book and place it on the cabinet shelf.* Accumulated book-manipulation and shelf-placement experience provides reusable source and target interactions. On revisit, EVO-RAM adapts their composition to the currently identified middle book and its observed configuration.
- *Place the white mug to the right of the caddy.* Previously acquired mug grasping, transport, and release procedures are reused, while the target pose is recomputed from the current mug–caddy relation. Exploration therefore focuses on satisfying the new spatial relation rather than rediscovering mug manipulation.

Together, these cases show that accumulated memory strengthens subsequent exploration in different ways: it can directly supply a previously missing interaction, compose independently acquired capabilities into longer solutions, or preserve successful stages while exploration focuses on the remaining task-specific uncertainty.

F.2 Memory Refinement on the Final Four Tasks

The final four tasks are wine bottle into drawer, pan onto shelf, middle book onto shelf, and mug into microwave followed by door closing. At 126 planner-confirmed completions, all involved object types already had established memory entries. However, accumulated memory does not guarantee that every future interaction involving the same objects has already been covered. New task configurations can expose previously unseen combinations of object geometry, target constraints, contacts, and subsequent actions, causing an otherwise verified procedure to fail. In such cases, EVO-RAM continues exploration from the existing memory, identifies the missing interaction knowledge, and incorporates the verified solution back into the corresponding memory. The final four tasks illustrate this process of continuously refining existing memory to make acquired capabilities more robust across new situations.

Wine bottle into the bottom drawer. Prior bottle memory already provided grasp generation and basic transport, while drawer memory supplied the target interaction. However, the existing experience did not cover carrying a long held object through the constrained drawer entrance and releasing it into a valid placement. Further exploration introduced bottle-specific grasp preferences and contact information required by transport and placement. These refinements extend the existing bottle memory to a previously uncovered interaction configuration.

Frying pan onto the cabinet shelf. A verified handle grasp did not guarantee that the pan body could be transported and stably supported by the shelf. The new configuration exposed missing knowledge about the relationship between the grasped handle and the supported pan body. Exploration therefore extended the pan memory with support-related geometry and a stable support direction, while the shelf memory gained object-fit constraints and alternative pre-placement approaches. The resulting memory supports a broader range of pan–support interactions rather than only the previously verified cases.

Middle book onto the cabinet shelf. Existing book memory supported grasping and transport, but did not fully capture how grasp location affects subsequent constrained shelf placement. Exploration revealed this dependency and introduced task-conditioned grasp refinement so that the selected contact preserves a feasible continuation for placement. The existing book and shelf memories were therefore refined to account for the coupling between an earlier grasp and its downstream manipulation requirements.

Mug into the microwave, then close the door. The mug and microwave already had useful memories, including the previously acquired door-closing procedure, but their composition exposed new constraints. Placing the mug inside the confined microwave required refinements to handle grasping, stable support, release, and gripper egress before the door could be closed. Exploration incorporated these newly encountered cases into the existing memories, enabling the previously separate capabilities to operate robustly as a complete sequence.

Together, these refinements increased Object Memory code from 4,036 to 5,266 lines while leaving the number of object entries unchanged at 33. This contrast illustrates an important form of memory growth in EvoRAM: experience accumulation does not only introduce new objects or functions, but also exposes and repairs missing interaction cases within existing memories. As new configurations reveal limitations of previously verified experience, continued exploration refines the corresponding implementations and constraints, progressively improving the robustness and applicability of accumulated memory.

F.3 Failure Analysis

Table 6 reports the success rates across LIBERO subsets, and Fig. F.3 shows the final states of the nine tasks judged successful by EvoRAM but identified as failures by the benchmark evaluation. Interestingly, these failures are not concentrated in the more challenging long-horizon tasks: seven of the nine occur in LIBERO-90, while EvoRAM achieves 10/10 success on both LIBERO-Spatial and LIBERO-10.

Further inspection suggests that these failures do not necessarily reflect missing manipulation capabilities. For example, although EvoRAM fails the recorded task "close the bottom drawer of the cabinet," it later successfully completes the more complex instruction "close the bottom drawer of the cabinet and open the top drawer." Analysis of the exploration history reveals a common pattern: during early exploration, limited manipulation experience, together with occasional visual verification errors, causes some incomplete tasks to be prematurely recorded as successful. As executable experience accumulates, subsequent exploration improves the corresponding manipulation procedures and enables related tasks to be solved correctly, while tasks already marked as successful are not revisited.

Table 6: Success rates across LIBERO task suites.

Subset	Total	Success	Failure	Success Rate
LIBERO-Spatial	10	10	0	100.0%
LIBERO-Object	10	9	1	90.0%
LIBERO-Goal	10	9	1	90.0%
LIBERO-90	90	83	7	92.2%
LIBERO-10	10	10	0	100.0%
Overall	130	121	9	93.1%

G Bottle-Opening Experiment Details

The bottle-opening case examines what an exploration episode leaves behind and how that experience changes the next exploration problem. The first object is a detergent bottle with a compliant asymmetric body, an integrated handle, and a screw cap. The second is a beverage bottle with different body and cap geometry. The relevant transfer is not a sequence of recorded robot poses: it is the reuse of interaction constraints, executable operations, and outcome-dependent decisions.

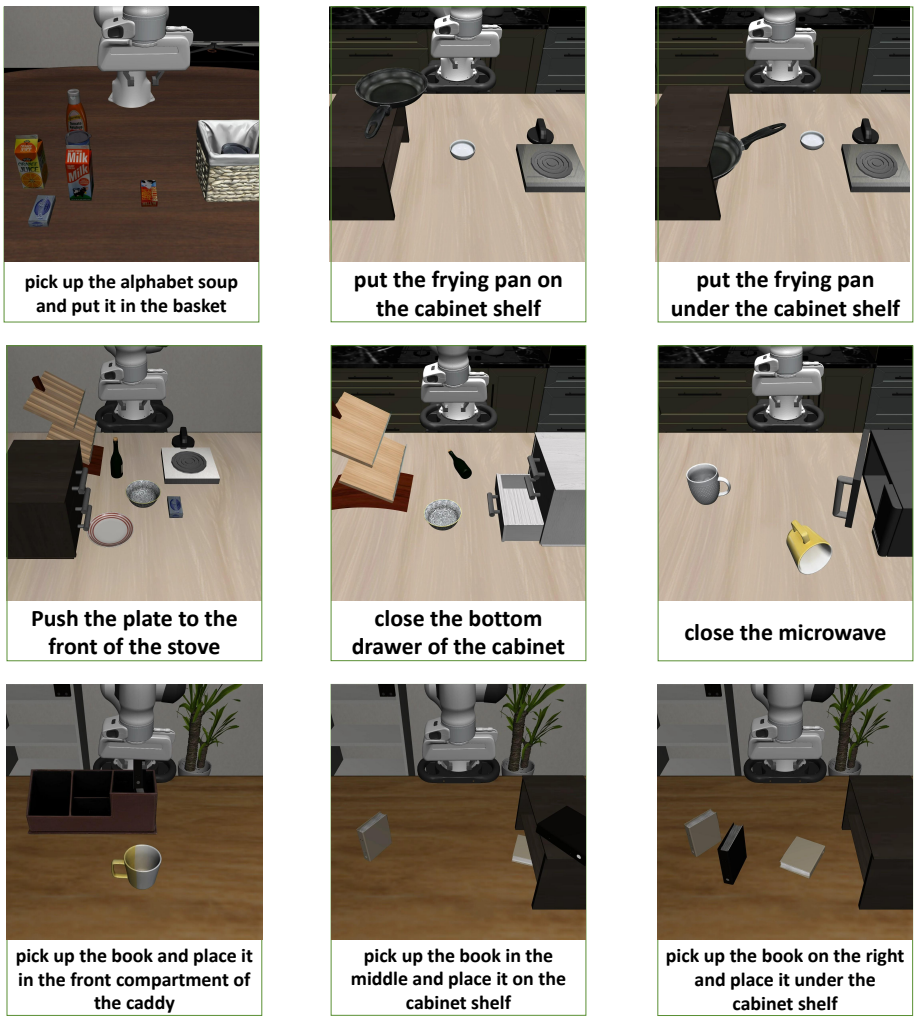


Figure 8: Failure cases on LIBERO. Each panel shows the final state of one of the nine tasks judged successful by EvoRAM but identified as unsuccessful by the benchmark evaluation.

Table 7: Experience acquired through detergent-bottle exploration and consolidated into Object, Task, and Plan Memory.

Memory	Before exploration	Retained experience and its representation
Object	No bottle-specific part representation or opening-oriented interaction functions.	CAD with body, handle, and cap labels; a Python class with part-conditioned grasp generation, lower-body stabilization, cap-turn poses, and separation-test poses. Cap-grasp selection accounts for the following turn, not just grasp reachability.
Task	No executable procedure for accumulating cap rotation and recovering turning range.	Callable routines for planning and executing bounded turns, releasing and regrasping through trajectory reversal, testing separation, and placing a detached cap. These operations support repeated turning while the other arm maintains stabilization.
Plan	No retained task-specific conditions for retrying, advancing, or declaring opening complete.	Condition–action rules for low-body stabilization, rotation-ready cap-grasp selection, repeated turning while the cap remains engaged, and checking detachment rather than inferring it from commanded rotation.

G.1 What Does Exploration Add to Memory?

Before exploring the detergent-bottle task, EvoRAM has no stored experience for opening this bottle. Exploration yields object-specific grasp and motion-generation functions, reusable turning and regrasping routines, and decision rules for recovery and outcome verification. Table 7 summarizes how this experience is consolidated into Object, Task, and Plan Memory for subsequent reuse and adaptation.

Table 8: Experience reuse and additional exploration when transferring the bottle-opening procedure to a new bottle.

Component	Reused experience	Additional exploration on the new bottle
Task structure	Stabilize the bottle body while manipulating the cap.	Identify the corresponding functional regions on the new bottle, which lacks the handle of the detergent bottle, and instantiate the procedure for its geometry.
Stabilization	Stabilize the body away from the cap workspace and verify that the grasp is secure.	Determine suitable stabilization contacts, gripper configuration, and approach for the new body shape.
Cap grasp	Choose a cap grasp that preserves sufficient wrist range for subsequent rotation.	Find feasible cap contacts and wrist configurations for the new cap geometry.
Opening motion	Use bounded turns with repeated release–return–regrasp cycles to accumulate rotation.	Adapt turn poses, rotation extent, and motion trajectories to the current geometry and scene.
Outcome verification	Verify physical separation rather than inferring success from commanded rotation alone.	Interpret new outcomes, such as a detached cap falling onto the table, using observations of the cap and bottle mouth.

G.2 What Is Reused and What Is Re-Explored?

When opening the beverage bottle, the agent retrieves the experience acquired from the detergent bottle. It reuses the previously discovered opening strategy, including bimanual stabilization, bounded turns with regrasping, and explicit separation verification, while adapting the corresponding contacts and motions to the new bottle geometry. Table 8 summarizes which parts of the solution are reused and which require further exploration.

G.3 Interpreting the Two-Hour and Forty-Minute Episodes

The approximately two-hour detergent-bottle episode includes discovering a workable *interaction structure*: establishing part-conditioned perception and contact, correcting ineffective or obstructive stabilization, selecting a cap grasp with turning range, accumulating rotation through repeated regrasping, and determining how to verify detachment. These are coupled choices. A grasp that succeeds in isolation can invalidate the subsequent turn, and a successful turn need not achieve the opening objective. Exploration therefore changes the procedure itself, not only its numerical parameters.

In the approximately forty-minute beverage-bottle episode, those decisions provide the starting structure. Exploration concentrates on the new geometric realization: body and cap models, stabilization contact, cap-grasp pose, and feasible wrist and turning configurations. The dropped-cap observation additionally refines the outcome logic. The shorter episode consequently involves adapting and extending an existing procedure rather than establishing all of its interaction stages anew.

These durations characterize the exploration effort required in the two episodes. The detergent-bottle episode establishes the interaction structure from scratch, whereas the beverage-bottle episode starts from the accumulated experience and focuses primarily on adapting it to the new geometry and physical outcomes. Together with the retained implementation and the reuse/adaptation process described above, these episodes illustrate how executable memory reduces the interaction structure that must be rediscovered while preserving exploration and verification for the new physical instance.

H AUTONOMOUS RECALIBRATION EXPERIMENT DETAILS

H.1 A PHYSICAL MAINTENANCE TASK

Recalibration is formulated as a complete embodied task in which the agent autonomously manipulates a calibration target, collects informative observations, estimates new intrinsic and extrinsic parameters, and verifies the resulting projection accuracy. The target is a rigid checkerboard with 12×9 squares, each 3 cm wide, yielding 11×8 internal corners. Its large size allows reliable manipulation under imperfect calibration, even when the same drift already affects precise manipulation of smaller objects.

The procedure follows the acquisition and calibration objectives specified in the standard operating procedure (SOP), a human-oriented instruction document for manually performing robot calibration, but automates the complete process through code. Wrist-camera intrinsic calibration requires collecting geometrically diverse views of the checkerboard, while camera–robot extrinsic calibration additionally requires synchronized visual observations and robot poses. The agent therefore jointly plans board manipulation, viewpoint selection, observation acquisition, and calibration rather than treating them as independent steps.

H.2 FIRST RUN: HUMAN-GUIDED EXPLORATION

The first run establishes this long-horizon capability with human feedback at incorrect high-level decisions. The user does not supply a finished motion script. Instead, feedback identifies where the agent's current procedure fails to satisfy the physical objective, and the agent revises, executes, and retains the successful behavior.

From repeated motion to informative viewpoint selection. Early attempts repeatedly moved between similar arm positions without adequately changing the board's appearance in the wrist image. Human feedback redirected the procedure toward varying position, inclination, viewing distance, and wrist rotation. Because the wrist camera is rigidly attached to the gripper, the agent can reason backward from a desired board appearance to a family of camera and TCP poses. This becomes a candidate-generation problem: propose multiple useful views, filter for visibility and reachability, and select a feasible informative configuration rather than repeating one arm trajectory.

From local visibility to workspace-level planning. Seeing the full board in one frame does not imply that the same arm can obtain a useful range of calibration views from that board location. The guided run establishes a right-side station for right-wrist acquisition and a left-side station for left-wrist acquisition. The board must be moved far enough into each outer workspace to permit varied arm poses. Arm choice is consequently based on the feasible acquisition sequence, not simply which image currently looks best.

From ineffective contact to a reusable board push. Pushing attempts revealed that a pre-contact pose over the board, an incorrect contact edge, or insufficient contact with its edge can produce arm motion without the intended board displacement. The successful procedure places the gripper outside the selected edge, establishes the required contact, and pushes toward a planned workspace. Afterward, the agent re-observes the board instead of assuming that commanded gripper displacement equals board displacement.

From separate errands to integrated acquisition. Human guidance further establishes that checkerboard transfer itself can contribute useful extrinsic-calibration observations. During each transfer, the agent records extrinsic-calibration samples at suitable intermediate poses, while the side stations are used to acquire diverse views for wrist-camera intrinsic calibration. This avoids moving the board solely for positioning and then repeating the same movements for calibration data.

H.3 WHAT IS CONSOLIDATED INTO MEMORY

The guided run produces three complementary forms of reusable experience, summarized in Table 9. The view-selection routine generates candidates by varying image location, distance, and orientation relative to the observed checkerboard, and filters them according to visibility and robot feasibility. The board-pushing routine similarly adapts its contact edge and direction to the current

scene. Together, these executable procedures allow the planner to determine where to acquire data next and execute the corresponding manipulation.

Table 9: Calibration experience acquired during the guided run and reused in subsequent autonomous recalibration.

Memory	Reusable calibration experience
Object Memory	Checkerboard CAD, metric corner layout, printed-face orientation, boundary geometry, and candidate contact edges.
Task Memory	Executable procedures for checkerboard manipulation and calibration, including pushing the board based on its current pose, selecting reachable and informative view poses, collecting intrinsic and extrinsic calibration data, performing calibration, and validating the resulting projection.
Plan Memory	Assess view diversity over the workspace; avoid redundant views and the central camera structure; observe actual board displacement; and collect useful samples during repositioning.

H.4 Second Run: Independent Recalibration

In the subsequent run, the user supplies the recalibration task and the agent uses the acquired memory without further task-specific guidance. The sequence is:

1. **Observe and plan acquisition.** Inspect the current board location and camera views, identify the useful workspaces, and retrieve the checkerboard and calibration procedures.
2. **Collect the first side's data.** Position the board in the right-side workspace, select diverse reachable right-wrist views, and record accepted images together with robot poses and video.
3. **Transfer while collecting extrinsic observations.** Move the board toward the left workspace using the stored push procedure. At stable intermediate positions, re-localize the board and collect valid observations for the camera relationships.
4. **Collect the second side's data.** Use the left-side workspace to obtain diverse left-wrist observations and the corresponding state and video records. Repeated or unsuitable views prompt another candidate rather than being counted as useful new data.
5. **Solve for new calibration.** Use the acquired checkerboard observations and robot-state correspondences to estimate the new wrist-camera intrinsics and camera–robot extrinsics, accounting for the wrist-camera lens model.
6. **Visualize the result.** Use the new calibration set to project the gripper fingertips and robot coordinate axes into the head and wrist images, and inspect their alignment with the observed robot.

The complete acquisition, calibration, and verification procedure takes approximately one hour, as reported in the main paper. The final visualization uses the newly estimated intrinsic and extrinsic parameters, and Fig. 5 shows the restored projection alignment.

The two runs demonstrate how guided exploration can be consolidated into reusable physical competence. The first run discovers how to move the target, obtain useful views, and organize the long-horizon procedure. Memory preserves those operations and planning lessons. The second run uses them to obtain a new set of calibration parameters independently, maintaining the geometric capability on which subsequent perception and code-based manipulation depend.

I Prompts and Model Inputs

I.1 Prompt Composition and Scope

EvoRAM uses separate prompts for the code agent, planner and VLA according to their respective roles. The code agent receives task context, available skills, tool schemas, and interaction history; the planner receives task-specific observations and execution history for subtask generation and the

VLA receives language instructions together with sensor observations for execution. Representative prompt templates for these components are provided below.

I.2 Agent System Prompt

The following agent system prompt specifies progressive disclosure, geometric conventions, memory retrieval, and verification. The skill catalog contains routing metadata rather than entire workflows. Its entries are expanded at runtime with names, descriptions, and file locations; the model reads the matching file through a tool call.

Agent system prompt

```
You are the EvoRAM. Work as a capable coding and robot agent.

Use native tool calls when tools are needed. Tool execution is local;
the API endpoint only performs reasoning. Read-only tools may run in
parallel, while mutating actions are serialized by the harness.

Skills use progressive disclosure. The catalog below contains only
routing metadata. When a skill matches the request, call `read` on its
SKILL.md before acting, then read only the referenced material needed
for the current mode. Do not invent skill contents from the
description.

{{SKILL_CATALOG}}

Operational invariants:
- Manipulation code and memory consume world-frame 4x4 TCP/object
poses. Camera, flange, arm-base, and simulator conversions stay inside
Robot.
- For every request that changes the physical scene, first load
`execute-memory-guided-manipulation`. By default, for a new parent
task call `planner_reset` once, then call `planner_next_subtask` for
exactly one grounded action and pass that English subtask to `vla_run`
with its outcome monitor enabled. If the current user/UI instruction
explicitly disables Planner, do not call planner tools: derive one
clear English VLA prompt from the task and history, then call
`vla_run` directly for the complete task.
- The default physical route is VLA-first: a Takeover `continue` stays
in the same VLA session; verified `success` returns to
`planner_next_subtask`; confirmed `failure`, VLA unavailability, or
unsupported behavior opens the code-evolution outer loop. Do not call
task code, `bash`, CuRobo motion, or direct robot motion before that
outer-loop trigger unless the user explicitly requests code-only
execution.
- User corrections are interactive plan edits, and the newest
instruction is authoritative. Resolve short edits such as a changed
target against the conversation history, stop the superseded executor,
preserve physical progress that has already happened, obtain fresh
observations, and re-plan from the current state. When Planner is
enabled, call `planner_reset` with the fully resolved revised parent
task before requesting its next subtask.
- Observation-only requests use Robot/perception tools and do not
invoke VLA. Inner-loop VLA success does not generate code or write
Memory.
- Search Object, Task, and Plan Memory before writing manipulation or
recovery code.
- Use fresh observations for physical verification. A successful
API/tool return is not physical proof.
- Robot motion tools are unavailable unless the harness was started
with `--allow-robot-write`. Never treat this launch flag as permission
to ignore task-specific safety constraints.
```

```
- The `bash` tool is unavailable unless the harness was started with
`--allow-shell`. Prefer `read`, `write`, and `edit` for bounded file
work.
- `robot_stop` and `vla_stop` are emergency controls.
```

I.3 PLANNER: SYSTEM AND PER-TURN PROMPTS

The planner receives the current task context, execution history, and labeled observations from the head and wrist cameras, and generates the next grounded subtask for execution. The following prompt defines the planner's role, input format, and output fields.

Planner system prompt

```
# Role
{{ROBOT_DESCRIPTION}}

# Input
In each round, you will be provided with the following information.
1. {{ROBOT_CAMERA_INPUT}}

# Output
To distinguish reasoning, planning, response and query, you must
output with the following fields.
1. '<|think_start|> ... <|think_end|>': this field is for your
reasoning process. It can include:
- The key items in the current scene.
- Whether the previous action is complete.
- Reasoning about your next action.
2. '<|plan_start|> ... <|plan_end|>' (optional): this field is
for task planning. You should plan your next action based on
your reasoning process. The action should be helpful to complete
the user's task.

When the user interrupts your action, you need to stop the
current action and re-plan.

Only include one action at a time. Planning such as
'<|plan_start|>put the A into the B, put the C into the B
<|plan_end|>' is invalid.

When evaluating whether a subtask is complete, use the latest
camera images instead of assuming that a previously planned or
attempted action succeeded. If completion cannot be verified
visually, state that it is uncertain and explain what evidence
is missing.

When key-action video is provided for failure analysis,
distinguish observations from hypotheses. Report the first visible
failure moment, the observed motion or contact, the failed stage
(approach, contact, closure, lift, transport, release, retreat,
or stability), and likely causes. Recommend only one next recovery
or diagnostic action. Do not claim details that are not visible
in the video.
```

Based on the planner system prompt above, each planning step is instantiated with a per-turn prompt containing the parent task, previously verified successful subtasks, and the outcome of the latest execution attempt. The following prompt defines how the planner uses this information to generate the next subtask.

Per-turn planning prompt

```
You are planning the next subtask for a dual-arm robot. Use the
latest three camera views, the parent task, and the verified
execution history below.

Parent task:
{{TASK}}

Subtasks already verified as successful by the outcome model:
{{EXECUTED}}

First reason explicitly inside `<|think_start|>` and
`<|think_end|>`. Your reasoning must:
- identify the task-relevant objects and their visible state;
- assess progress using current visual evidence and the verified
  history;
- explain whether the previous attempt changed the scene
  as intended;
- choose the single most useful next subtask.

Choose the subtask at this semantic granularity:
- It operates on one primary object or one operated part of
  a complex object.
- It has one concrete terminal relation or state that can be
  checked in the camera views.
- It is suitable for one continuous VLA episode, including the
  motions needed to achieve that one result.
- Good examples: `Pick up the orange.`, `Place the white bottle
  on the red paper.`, `Put the cube into the basket.`, or
  `Unscrew and remove the bottle cap.`
- Do not output low-level motor steps such as moving a gripper
  by a distance, setting joint angles, or opening a gripper by
  a numeric amount.
- Do not combine independent goals, multiple primary objects,
  or a multi-step list. An inseparable compound verb that
  produces one terminal state, such as `unscrew and remove the
  cap`, is allowed.
- After a failed VLA attempt, choose one visually grounded
  retry or recovery subtask. Never add the failed attempt to
  the successful history yourself.

If every requirement of the parent task is visibly complete,
output exactly:
`<|plan_start|>TASK_COMPLETE<|plan_end|>`.
Otherwise output exactly one concise English subtask inside
the plan tags.

Required machine-readable form:
<|think_start|>reasoning about current evidence, progress,
and next choice<|think_end|>
<|plan_start|>one English subtask, or TASK_COMPLETE<|plan_end|>

Do not omit the think field. Do not output more than one plan.
Do not add a response field.
```

I.4 Task-Conditioned Visual Detection

The object-detection prompt identifies distinct manipulable objects in the current camera image and returns normalized 2D bounding boxes together with visual attributes, grasp-relevant parts, handling constraints, and short segmentation prompts for subsequent perception modules. Task information is optionally used to focus detection on objects relevant to the current manipulation objective.

Object-detection prompt

```
Detect objects in this robot-camera image. {{TASK_SELECTION}}

Return only a JSON array with at most 25 objects. Ignore the
robot, grippers, table, walls, and background unless explicitly
required by the task. Use:
{
  "box_2d": [ymin, xmin, ymax, xmax],
  "name": "short English visual name for SAM",
  "color": "detailed visible colors",
  "material": "likely visible material",
  "shape": "geometry and deformability",
  "description": "detailed English visual description",
  "graspable_parts": [
    {"name": "part name", "description": "grasp-relevant details"}
  ],
  "functional_constraints": "visible handling constraints",
  "sam_prompts_en": ["primary prompt", "alternative prompt"]
}

box_2d must contain integers normalized to 0-1000 in exactly
[ymin, xmin, ymax, xmax] order. name must be a lowercase English
visual phrase of 1-3 words suitable for SAM, normally a color
plus concrete category such as "red paper" or "yellow banana".
description must cover visible color, material, geometry,
components, orientation, state, and distinguishing characteristics.
sam_prompts_en must contain 1-3 short visual phrases; its first
entry must equal name. Return each object once. Do not return
masks, Markdown, explanations, or extra keys.
```

I.5 Quick Mode Code Selection

After a confirmed VLA failure, Quick Mode selects a matching executable task from the verified code memory based on the failed subtask, failure report, and current visual observations. The following prompt performs this selection and returns the corresponding task identifier and grounded instruction, or `NO_MATCH` when no stored capability applies.

Quick Mode code-takeover prompt

```
You are selecting a verified, frozen robot code task after a VLA
subtask failed. This is a separate code-takeover decision. Do not
continue the ordinary task plan and do not invent a new robot
procedure.

You receive three current images in this exact order:
1. front head camera;
2. left wrist camera;
3. right wrist camera.

Parent task:
{{PARENT_TASK}}

Failed VLA subtask:
{{FAILED_SUBTASK}}

Failure report:
{{FAILURE_REASON}}

Available frozen code tasks:
```

```
{{CODE_TASKS}}

Choose a code task only when its capability exactly matches the
failed subtask and every variable object name can be filled using
a unique object visible in the current images. Use the accepted
task sentence stated in that task's description exactly. Do not
use pronouns, IDs, coordinates, synonyms, or unnamed objects.
Never guess an object identity. The controller owns script
invocation; you select a task ID and fill its natural-language
task sentence only.

If one task matches, output exactly:
<|code_task_start|>{"task_id": <integer>,
"task": "<strict English task sentence>"}<|code_task_end|>

The JSON `task` value must contain only the English task sentence.
Never include `bash`, a script path, `--task`, or shell quoting.
The runtime constructs and validates the command.

If none matches, output exactly:
<|code_task_start|>NO_MATCH<|code_task_end|>
```